PROGRAM ANALYSES FOR RESILIENT AND APPROXIMATE COMPUTATION

by

Arvind Haran

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2016

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Arvind Haran**

has been approved by the following supervisory committee members:

Zvonimir Rakamarić	, Chair	**12/18/2014** Date Approved
Ganesh Gopalakrishnan	, Member	**12/18/2014** Date Approved
Rajeev Balasubramonian	, Member	**12/18/2014** Date Approved

and by **Ross Whitaker** , Chair of

the School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

To minimize resource consumption and maximize performance, computer architecture research has been investigating approaches that may compute inaccurate solutions. Such hardware inaccuracies may induce a wide variety of program behaviors which are not observed when the program executes on reliable hardware. Future programmers will need to reason about these different behaviors in order to achieve efficient resource utilization without compromising on error resilience. To ensure resilience, critical computations need to be protected from the potentially fatal effects of hardware faults. On the other hand, to exploit the resource-accuracy trade-offs, certain computations need to be approximated to a tolerable extent. Effective techniques that provide insights into inaccuracy-induced changes in application behavior are required to achieve these competing objectives.

In this thesis, we propose dynamic and symbolic program analysis techniques to profile the inherent resilience of applications. At their core, both approaches analyze the effect of a targeted injection of an inaccuracy on the behavior of an application in comparison with the pristine execution. In the dynamic-analysis-based approach, the comparison is performed by a tool that injects runtime inaccuracies into an application. The tool probabilistically injects faults, with bit-level granularity, so as to simulate execution in an environment that is prone to single or multibit faults. We provide the results of a comparative study of sorting algorithm resilience using this approach. In our second approach, we provide a symbolic control flow differencing technique to reason about control flow behavior of applications that are subject to inaccuracies. Particularly, we define a precise symbolic encoding of control flow equivalence using uninterpreted functions. We also provide an evaluation of a prototype implementation on several benchmarks, showing promising results.

Dedicated to my Parents and Teachers

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Before we get distracted by the details of the work that I've done over the course of my masters program, I'd like to, more importantly, express and emphasize my gratitude to all those who have been integral for this 2+ year learning process.

First of all, I thank my parents for all their support and for being an undepletable source of unconditional love.

My advisor, Dr. Zvonimir Rakamarić, is the most patient, understanding, and smart person with whom I've worked. I thank him for being an embodiment of all the qualities of a great advisor and for all the guidance, motivation, and support during the course of completion of this thesis. Working under his mentorship has truly been a thoroughly enjoyable learning experience.

I thank my committee members, Ganesh Gopalakrishnan and Rajeev Balasubramonian, for their suggestions and feedback, which have helped me improve this work. I thank Prof. Ganesh specifically for his support during the initial months of my masters program.

I am grateful to Shuvendu Lahiri, who has been extremely kind and has selflessly helped me with all the problems I faced during my research, despite his busy schedule. I thank Vishal Sharma for his support and for being a great teammate when we worked on KULFI. It was a truly enjoyable team effort. I am thankful to Prabhakar Kudva, Karthik Pattabiraman, Pedro Diniz, Nathan DeBardeleben, and Greg Bronevetsky for all their invaluable and timely inputs for my research.

Last but not least, I would like to thank Karum, Prashanth, Pandeyji, Paanpaas, Sidharth, Chotus, Arun, Naveen, Praveen, Shashank, Anand, Sriram, and Ann, for the great company and fun times. I especially thank Arun, Naveen, and Praveen for teaching me some courses in their own unique styles, from which I derived more than the lectures themselves.

This work was supported by the SRC grant *2426.001 Localized, Layered, Formal Hardware/Software Resilience Methods.*

PREFACE

I conducted the research presented in Chapter 2 of this thesis in collaboration with my MS advisor Dr. Zvonimir Rakamarić, committee member Dr. Ganesh Gopalakrishnan, and colleague Vishal Sharma. This work was also originally published with them as coauthors [1]. Much of the original text of this publication has transitioned into the thesis. As a result, some of the sentences and paragraphs of that chapter were authored by Zvonimir, Ganesh, and Vishal.

In Chapter 3, I describe my research on profiling application resilience using symbolic differencing. This work has been done in collaboration with Dr. Zvonimir Rakamarić and Dr. Shuvendu Lahiri from Microsoft Research. Most of the text in Chapter 3 comes from a manuscript of this work which is in preparation for a publication. Therefore, although the majority of the work was done by me, there are sentences and paragraphs in that chapter that were authored by Zvonimir and Shuvendu.

CHAPTER 1

INTRODUCTION

The increasingly important concerns of power and energy consumption, along with ambitious performance goals, have resulted in methodologies in computer architecture research that may compute inaccurate solutions. Depending upon the underlying application, such inaccuracies may either exhibit fatal program behaviors (such as crashes, silent data corruptions that cause unacceptable anomalous behaviors, etc.) or a range of program behaviors with reasonable accuracy bounds. These inaccuracies may either be caused by unreliable hardware, or introduced intentionally by developers for achieving performance and energy gains on supporting platforms.

With the growing scale of systems and the level of integration of transistors within CPU (Central Processing Unit) and GPU (Graphics Processor Unit) cores, undetected bit-flips pose a serious challenge to our ability to rely upon computational results [2, 3, 4]. Recent studies [5, 6] show that it is unaffordable to employ hardware-only solutions to detect (and hopefully correct) hardware faults. A follow-on study [7] in fact expresses the need for software-based solutions to be used in tandem with hardware-based solutions. One may think of these software solutions as monitors placed within the code to trap errors. The behavior of an application subject to transient bit-flips may resemble the behavior of the same application with some random logical bugs. The programmer requires insight into the behavior of an application under transient faults to address the problem of synthesizing effective monitors to trap unacceptable behavior.

Many computations, however, can tolerate occasional runtime inaccuracies. *Approximate computation* is a novel paradigm for a particular class of applications that have an inherent trade-off between accuracy, performance, and power [8, 9]. Examples of applications that handle inaccuracies with a level of tolerance include multimedia, machine learning, big data, and financial computations. The approximate computation paradigm identifies the possibility to adapt some computations in such applications to deliver inaccurate results in exchange for relaxed correctness, while consuming substantially less power or compute time. Recent

frameworks [10, 11] have equipped programmers with constructs to *manually* annotate source code and data, which then the underlying system would use to approximate computations. Thus, it is up to the programmer to annotate the program with care to avoid under-, over-, and fatal approximations, and to achieve optimal energy savings.

Current techniques force programmers to manually infer the resource-accuracy trade-offs, which is difficult, time consuming, and unreliable. Previous work [12] on application-specific DRAM (Dynamic Random Access Memory) power-management identifies the problem of determining criticality in applications where the critical state is tightly intertwined with the noncritical state. Incorrectly labelling objects as noncritical would cause the corruption of a critical state leading to application failure, whereas failure to identify noncritical regions would lead to lost opportunities in power savings. The labelling approaches are, typically, conservative since preventing application failure is more critical than resource-efficiency. To provide such safe annotations and place better error detectors during the development phase, the programmer needs to understand the changes in properties of the approximated application with respect to the precise version. This thesis is geared towards exploring dynamic and symbolic strategies to analyze the resilience applications and enable automation of such program annotations, thereby providing the programmers with tools that help them write resilient and efficient code.

1.1 Thesis Statement

We hypothesize that dynamic and symbolic analysis of deviant behaviors induced in an application, by targeted injection of inaccuracies, can be used to profile application resilience, therefore enabling resilient and approximate computation.

1.2 Thesis Contributions

Our main contributions are summarized as follows:

1. Dynamic analysis of applications using the KULFI fault injector.

   (a) We present a new compiler-level fault injector called KULFI that conveniently simulates faults occurring within CPU state elements. We describe how the features of this (publicly released [13]) tool compare with existing compiler-level fault injectors.

   (b) As a case study, we run sorting algorithms under controlled fault-injection scenarios and empirically observe their behavior in order to determine instructions

critical to application resilience. We plot the faulty behaviors observed and provide evidence that our results are statistically significant.

2. Static symbolic analysis for control flow differencing.

   (a) We formalize the notion of a fault affecting the control flow behavior of a program.

   (b) We encode this problem into an instance of a program equivalence verification problem.

   (c) We provide a precise symbolic encoding of control flow equivalence using uninterpreted functions.

   (d) We implement a prototype tool for control-flow-based profiling of application resilience.

1.3 Thesis Overview

In Chapter 2, we describe our approach to using dynamic analysis to profile application resilience and the anatomy of the tool, KULFI, that performs the fault injection necessary for the analysis. We also present the results of the case-study of comparing a family of sorting algorithms on their resiliency. We conclude the chapter with a discussion on the advantages and disadvantages of using dynamic analysis-based techniques for profiling application resilience. In Chapter 3, we describe our symbolic differential program analysis approach to detect program components that cause a control flow divergence in the presence of inaccuracies. We also provide our encoding for precise and efficient control flow differencing using uninterpreted functions and a comparison with traditional approaches such as taint analysis. We discuss previous work related to dynamic and symbolic analysis for resilience in Chapter 4. In Chapter 5, we provide our concluding remarks and outline of future work.

CHAPTER 2

DYNAMIC ANALYSIS FOR RESILIENCE

Previous studies [14] indicate a trend of increasing failure rates in the circuit level due to factors such as dynamic variations in supply voltage, temperature, aging and radiation. Resilience has been quoted to be one of the major roadblocks for HPC (High Performance Computing) in future exascale systems as multiple failures are expected every day [15]. Fault-tolerant computing has been very actively researched for decades, and forms the basis of many practical techniques in use, including redundant designs, checkpointing, voting schemes, and hardware-level error and correction schemes [16, 17, 18]. There have been extensive studies to characterize the effect of faults, occuring in the circuit level, on programs implemented in hardware and software. [19]. Fault injection has been a common methodology to evaluate the dependability of applications [20]. Our first approach to profile resilience of a given application is based on fault injection.

In this chapter, we first define the common terms of studies on resilience followed by the architecture of our fault injection tool. In order to show the applicability of our dynamic analysis to infer the resilience characteristics of applications, we conclude the chapter with the setup and results of a case-study on sorting algorithms performed using the outlined fault injection tool.

2.1 Preliminaries

We first define the common terms used by resilience studies in literature [4, 21]. The rest of this document is consistent with these definitions. The manifestation of faults at the software level can be modeled by *flips* (changes) in bit-values of the computational state. Depending on the nature of these state changes, one can classify faults as follows.

1. *Permanent Faults:* Permanent faults are those that, once introduced into a state element, persist for the remainder of the computation, thus modeling permanent hardware failures.

2. *Transient Faults:* Transient faults are those that may disappear as well as reoccur during a computation. Since transient faults model rare events, such as alpha particle strikes or marginal circuit operation (often caused by noise), it is customary to study a given computation under a *single* transient fault occurrence.

When a fault occurs, the effect can be one of the following, as captured by fault filtration that occurs across the hardware/software stack [5]:

1. The fault falls within the microarchitectural don't-care set, thus effectively getting filtered.

2. The fault reflects as a visible microarchitectural state effect, but is filtered by the instruction set architecture (ISA), for example by being over-written by a good value at the beginning of the next microarchitectural epoch.

3. The fault is reflected into a programmer visible register, but falls within the don't-care set of the application logic, say by affecting a variable that does not form the "answer" returned by a function call.

4. The fault causes the machine to hang, results in a segmentation fault, or is otherwise clearly observed (say, by tripping a built-in hardware-level error detector).

5. The fault silently corrupts the output of a computation without tripping any observer or without being filtered.

We will use the term *Benign Fault* for categories 1–3 and *Silent Data Corruption (SDC)* for category 5. We will assume that *Segmentation Faults* are the only observed category 4 of faults.

2.2 KULFI: A Fault Injector

We have developed an open-source instruction-level fault injector named Kontrollable Utah LLVM Fault Injector (or KULFI)[1] on top of the LLVM (Low Level Virtual Machine) compiler infrastructure [22, 23]. KULFI [1] is capable of injecting static and dynamic faults into programs written in C. Static faults model permanent faults and are injected into a fault site selected during compile time. Dynamic faults emulate transient faults and are injected into a fault site selected during program execution. KULFI can inject faults into both data and address registers, and currently it models only single-bit faults. The fault sites could be

[1] Available from `http://github.com/soarlab/KULFI/`.

viewed as processor registers containing data or address mapped to a program state. This provides fine-grained control over the fault injection process by allowing a user to specify fault injection probability, injected byte location, fault site type (data, address, or both), limit on the number of injected faults, target functions to inject faults into, etc.

Figure 2.1 shows the flowchart of the dynamic fault injection done by KULFI. At a high level, the fault injection loop goes through all dynamic instructions. A dynamic instruction is an instance of a static instruction at runtime. Note that there may be several dynamic instances of a single static instruction due to the control flow characteristics of the program. For each dynamic instruction, a type of fault to be injected is selected as either a data or pointer fault type. First, KULFI randomly selects a C function for fault injection from a set of user-defined target C functions. Then, it randomly selects from the chosen function an instruction for fault injection, and determines (based on user input) whether the fault type for the selected instruction would be a data or pointer fault type. Subsequently, KULFI checks whether it is feasible to inject a fault with the chosen fault type into the selected instruction. If this check passes and the provided fault injection probability is met, then a fault is injected into the instruction. After successfully injecting a fault, KULFI checks whether the preset limit on the number of faults has been reached. These steps are repeated for all dynamic instructions. Once the loop is finished going through all the dynamic instructions, the execution of KULFI terminates. If the end of the dynamic instruction sequence is not reached, the fault injection loop is repeated; otherwise, the execution of KULFI terminates.

Given that transient faults are the main focus of this work, we describe in more detail how KULFI models such faults. Figure 2.2 illustrates a transient fault occurring at register level. The shown register does not contain a fault at time t_1. At time t_2 a fault occurs, and then it disappears at time t_3. Dynamic fault injection capability of KULFI models such transient fault behavior. KULFI operates on the LLVM intermediate representation (IR) level (i.e., LLVM bitcode level) in the static single assignment (SSA) form [24]. SSA ensures that every IR variable (i.e., logical register) is assigned only once, which is an advantage as opposed to operating at the source code level when modeling transient faults. More specifically, injecting a fault into an SSA logical register referenced by an instruction is a one-time occurrence affecting only the instructions that use that logical register. SSA naturally prevents references to the same source code variable in the later instructions from observing the injected fault. Note that the duration for which a transient fault persists in an actual hardware register varies. Therefore, it is possible that more than one instruction could get affected from a single transient fault. Currently we do not capture such timing-related behaviors of transient faults in the software emulation of faults done by KULFI. However,

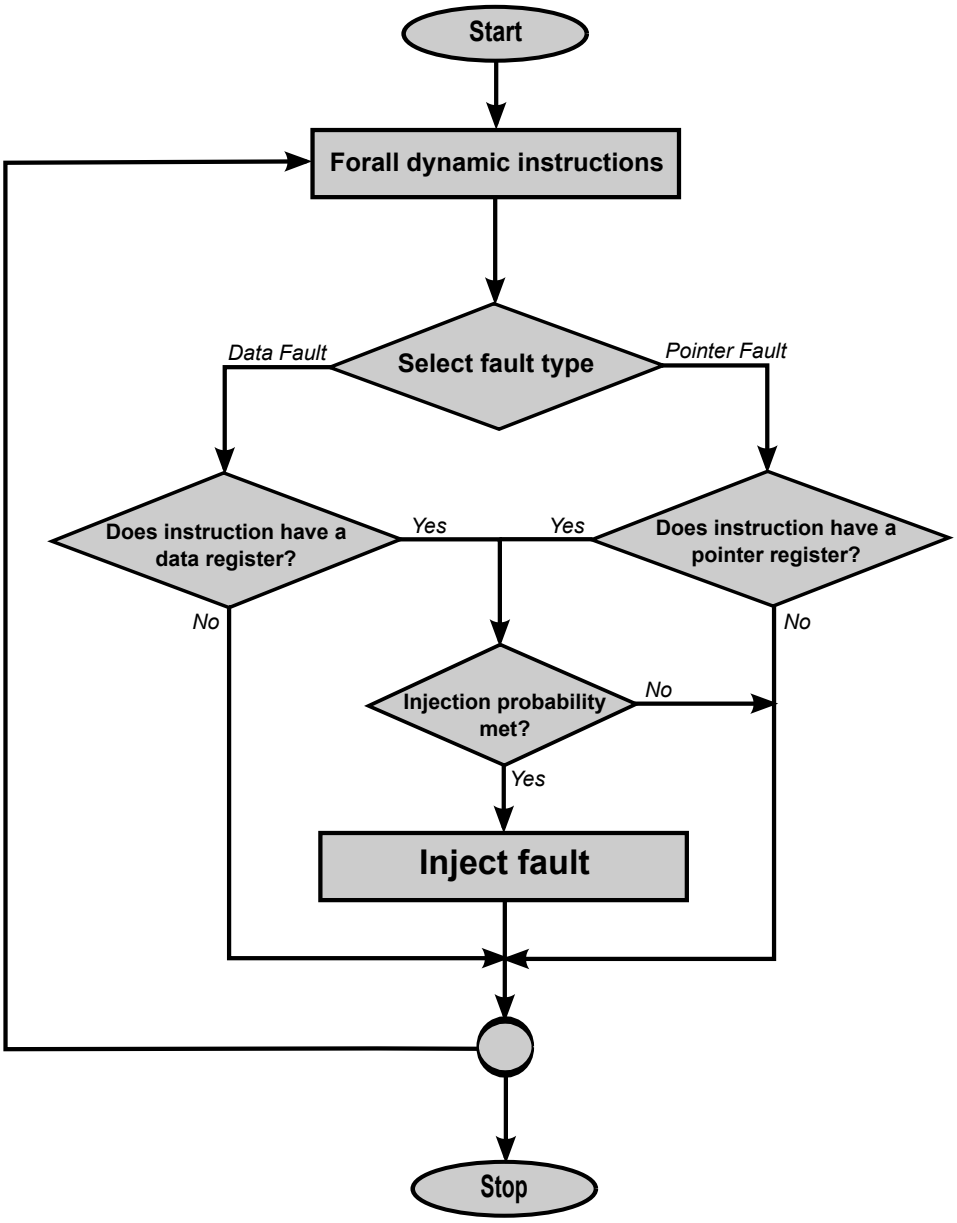

Figure 2.1: Flowchart of Dynamic Fault Injection in KULFI

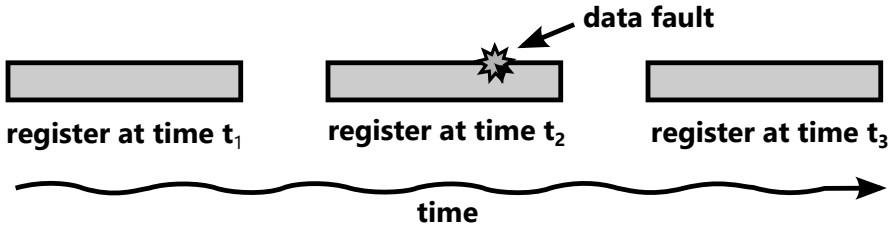

Figure 2.2: Transient Fault Occurring in a Register

KULFI still provides a reasonable model of transient fault behaviors, and similar models were adopted by other error injectors [25, 26].

2.3 An Empirical Case-study of Sorting Algorithms

We have performed an empirical case study that assesses the resilience of several popular sorting algorithms. Previous work suggests that algorithms and data structures that solve a particular problem not only vary in time and space complexity, but also in how resilient they are to faults [27]. In that line of work, a memory (i.e., DRAM) fault model is assumed to study the resilience of sorting algorithms [28]. However, the memory fault model is often too restrictive since it fails to cover classes of faults not directly tied to memory, such as register corruptions, control flow corruptions, and incorrect computation, which are prevalent in real-world systems. In our empirical study, we choose to use a more expressive fault model supported in KULFI. The chosen fault model considers all instructions of a program as candidate fault occurrence locations, including memory reads and writes, register operations, and control flow instructions.

In our case study, we consider implementations of five well-known sorting algorithms: BubbleSort (with preemptive termination criterion), RadixSort, QuickSort, MergeSort, and HeapSort.[2] All implementations take as input an array of integers to be sorted, and they output the sorted array. Since this is a preliminary study, we do not bias on the size and input data, i.e., the arrays are of random size (between 2000 and 10000) and contain random integer data. (As part of future work, we plan to experiment with various fixed data sizes and algorithm-specific inputs.)

We perform a *fault injection campaign* for each sorting algorithm implementation using KULFI. Each fault injection campaign consists of 200 *fault injection experiments*. A single fault injection experiment is comprised of 100 executions of an algorithm. Therefore, each algorithm is executed a total of 20000 times, which we split into 200 fault injection experiments so that we can later compute the statistical significance of our results. In each execution, the algorithm operates on a different randomly generated input array, while a single random bit-flip error is injected at runtime using KULFI. We describe the details of our fault injection strategy next.

---

[2]Source code of the examples and scripts for performing the experiments are also available from the KULFI website.

2.3.1 Fault Injection Strategy

Even with a fault injector such as KULFI available, selecting a realistic fault injection probability requires careful planning; we now present our approach in this regard. As noted in Section 2.1, fault filtration naturally occurs across the hardware/software stack where many faults fall into the "don't-care" sets of the higher layers. Specifically, Sanda et al. [5] report how an IBM POWER6 processor was actually bombarded with protons and alpha particles within an elaborate experimental setup. The authors estimated that the percentage of faults that actually reached the application logic was 0.2% of the overall number of latch-level faults. While this approach to fault simulation is quite realistic, such "bottom-up" fault injection approaches (and its infrastructural overheads) are clearly out of reach to most researchers. On the other hand, there are a number of recent approaches targeting software-level resilience enhancing mechanisms (see Chapter 4). Therefore, we decided to focus our empirical study only on the effects of faults that *do reach* the application logic since those are of a particular interest to the software-level resilience community. We still had to devise a reasonable and fair fault injection probability.

Given the above discussion, we now define additional notions that help us elaborate our studies. By the term *dynamic instruction* we refer to a runtime instance of a static LLVM program instruction. We define the *dynamic instruction count* as the actual number of dynamic LLVM instructions executed corresponding to a specific program execution. For example, for a simple program consisting of five static instructions in a loop that iterates 1000 times, the static instruction count is five, while the dynamic instruction count is 5000. For our sorting algorithms, the dynamic instruction count varies depending on the algorithm considered and the input array, which we have to take into account to ensure that all dynamic instructions are considered for fault injection with equal probability. Table 2.1 gives various statistics for our sorting algorithms:

- LOC is the number of lines of code,
- SIC the number of static fault site instructions,

Table 2.1: Statistics of sorting algorithms

Algorithm	LOC	SIC	MinDIC	MaxDIC	AvgDIC
BubbleSort	56	13	68k	61442k	14818k
RadixSort	61	39	30k	2040k	565k
QuickSort	65	25	34k	1110k	303k
MergeSort	70	38	79k	1269k	364k
HeapSort	77	28	15k	1519k	500k

- MinDIC the minimum dynamic instruction count,
- MaxDIC the maximum dynamic instruction count, and
- AvgDIC the average dynamic instruction count.

We initially perform a faultless run of an algorithm on an input to compute the dynamic instruction count N for a particular execution. We then define the probability of fault injection for each dynamic instruction to be $1/N$. This ensures that all dynamic instructions are equiprobably considered for fault injection in subsequent runs of the program on the same input. Such a fault injection strategy models real-life faults as close as possible at this level of abstraction.

Figure 2.3 illustrates our fault injection strategy for this case study performed using KULFI. First, a sorting routine is compiled using LLVM's C/C++ front-end Clang into an LLVM bitcode file, which contains LLVM's intermediate representation. Then, we execute the generated bitcode file using the LLVM virtual machine (i.e., *lli*) and as input we provide a randomly generated input array. We record the sorted output array for later comparison. In the process, we also measure the dynamic instruction count N for this particular faultless execution. Using the dynamic instruction count, we compute the probability of fault injection for each dynamic instruction as $1/N$. The original LLVM bitcode file and the computed fault injection probability are given as inputs to KULFI. The tool generates a fault-injecting LLVM bitcode file, i.e., an instrumented version of the original bitcode file in which a transient fault might be injected during execution into a dynamic instruction with the computed probability. The fault-injecting LLVM bitcode file is then executed on the same input array. We observe the number of injected faults and log only the executions during which exactly one fault is injected; we call such executions 1-fault executions. Executions where the number of injected faults is not equal to one are discarded. We record the outcome of every 1-fault execution to later analyze the effect of fault injection.

2.3.2 Experimental Results

We identify three possible outcomes of an execution of a sorting algorithm with a fault injected at runtime.

1. **Benign Fault.** In general, a transient fault is benign when the program state at the end of a faulty execution is the same as the program state obtained after a faultless execution. In the context of sorting algorithms, the output array obtained as the result of a faulty execution has to exactly match the sorted output array of the faultless execution.

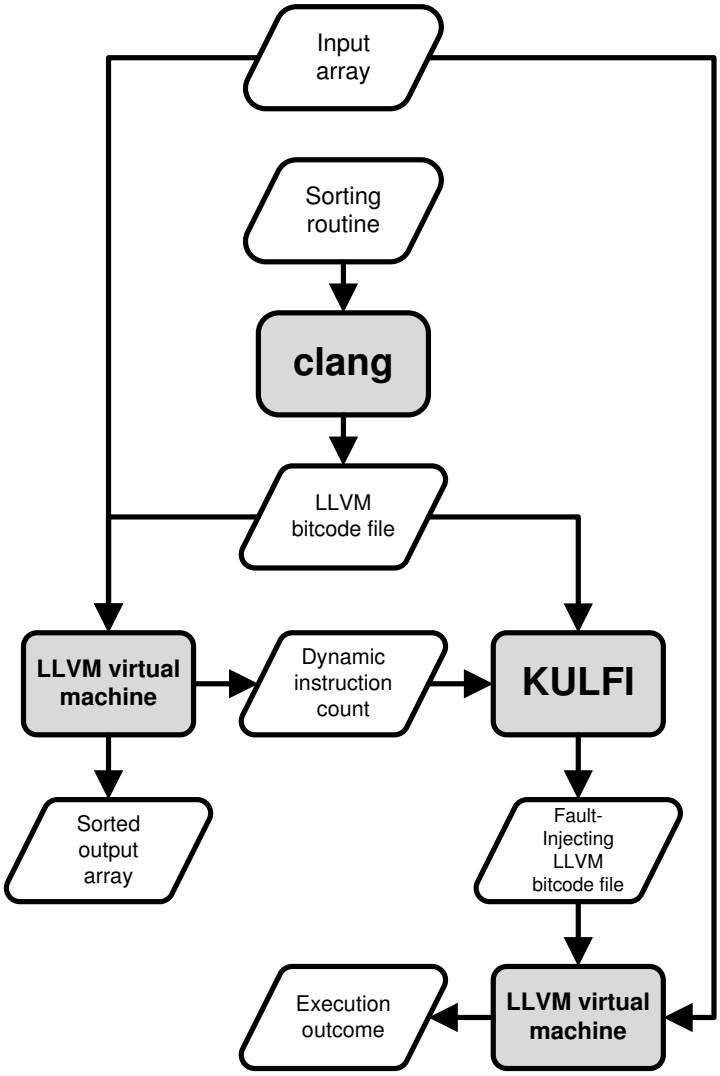

Figure 2.3: Fault Injection Strategy

2. **Segmentation Fault.** We classify a transient fault that causes a program to crash due to performing an invalid memory access as a segmentation fault.

3. **Silent Data Corruption (SDC).** A fault is classified as an SDC when the ordering, frequency, or the set of array elements at the output of the faulty execution is different from that of the faultless execution.

After each fault injection experiment (i.e., 100 1-fault executions), we log the number (i.e., fraction) of executions falling into each category. For example, here is how one such log entry might look:

```
Benign: 41, Segmentation: 29, SDC: 30
```

In the end of every sorting algorithm fault injection campaign, we are left with 200 such log entries, one for every fault injection experiment. We perform statistical analysis of these logs and present our empirical results next.

From Figures 2.4–2.6, we observe that the values in log entries obtained after every fault injection campaign are strongly clustered, and there is a statistically significant distribution of the fractions for each outcome category. The larger shapes (e.g., triangles) in the middle of clusters are indicative of the larger number of instances of faults that are closer to the middle. More specifically, the fractions of every category of outcomes across the 200 fault injection experiments follow the 68-95-99.7 (or three-sigma) rule of normal distribution. Therefore, using our empirical data, we can draw statistically significant conclusions about the behavior of the analyzed sorting routines in a faulty environment.

Figure 2.7 details the comparison of the sorting algorithms based on the average number of executions in each category of fault outcomes. For example, we observe that BubbleSort, though an algorithm with higher time complexity than HeapSort, leads to more detectable faults. In fact, HeapSort is the least resilient with respect to SDCs and results in either benign faults or SDCs in its fault injection campaign. QuickSort masks the majority of injected faults and therefore its high number of benign faults. It is worthwhile to note that the three algorithms that have the least number of detectable faults (MergeSort, QuickSort, and HeapSort) follow a recursive divide-and-conquer algorithm design paradigm. On the other hand, BubbleSort leads to more segmentation faults.[3] We observe that approximately 85% of the executions of QuickSort and 90% of the executions of BubbleSort avoid SDCs. To

[3]As to the reliability and repeatability of measuring segmentation faults, one has to choose virtually identical runtimes and memory layouts as well as mappings of user variables to memory locations. We will address these considerations in future work.

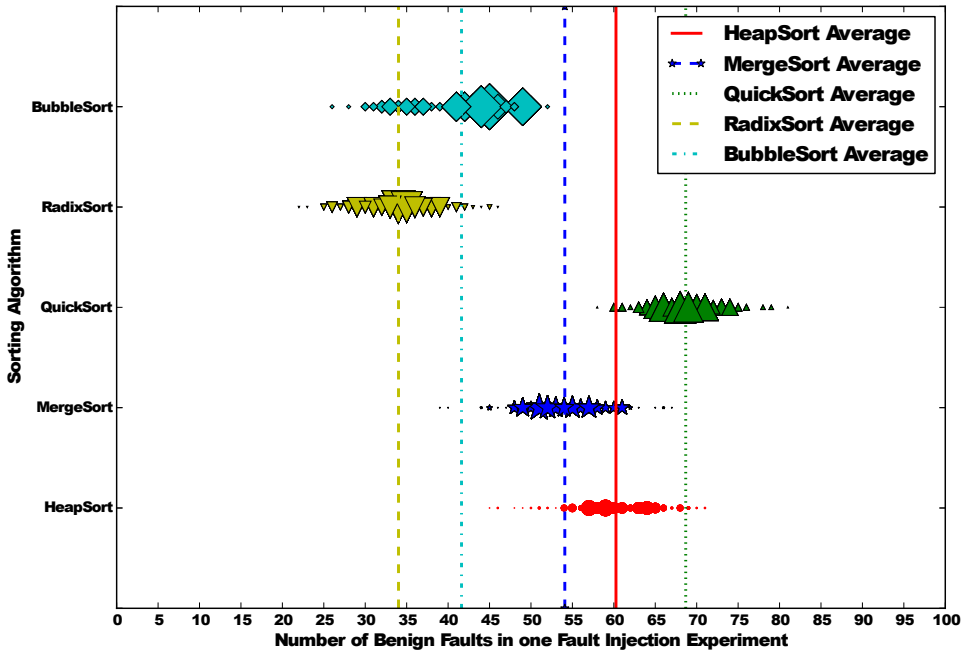

Figure 2.4: Benign Faults

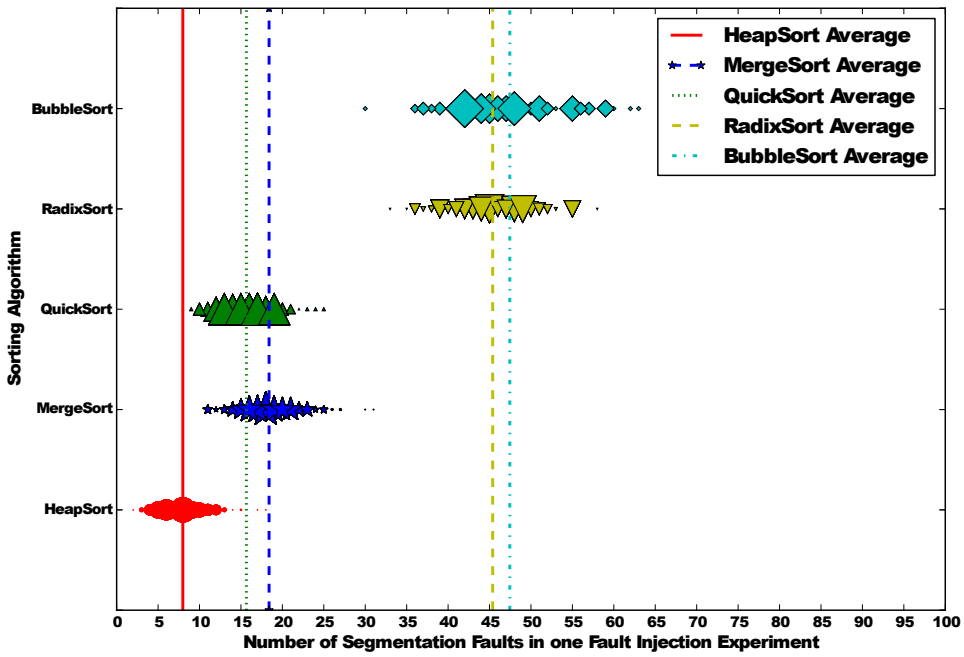

Figure 2.5: Segmentation Faults

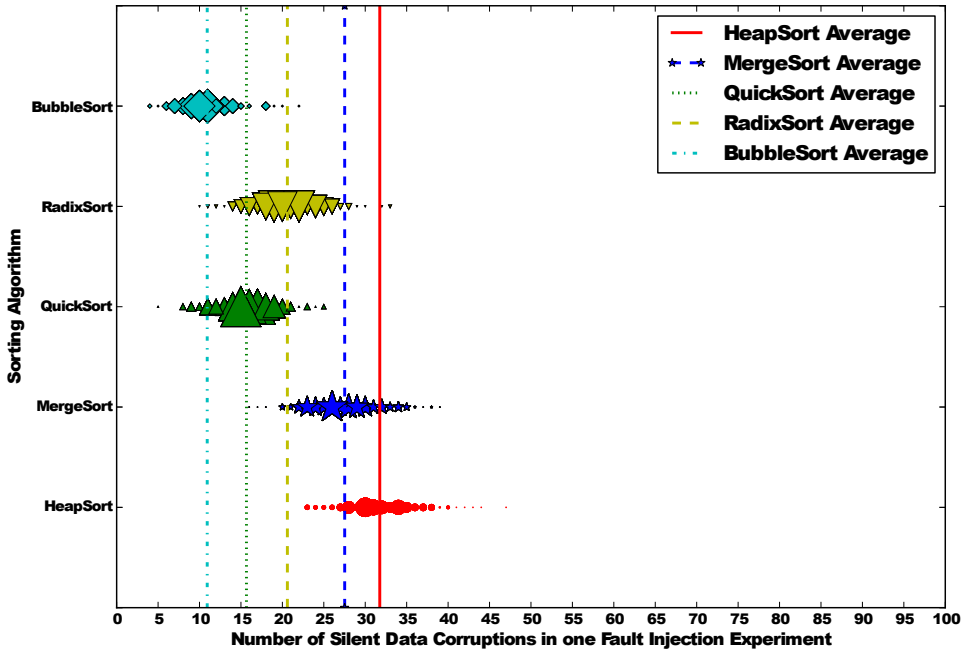

Figure 2.6: Silent Data Corruptions

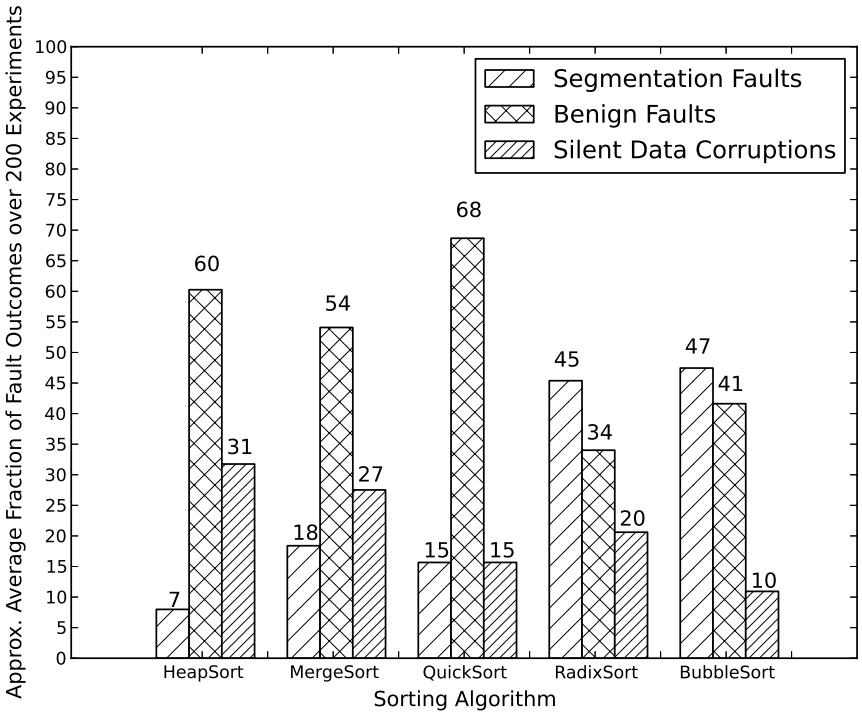

Figure 2.7: Summary of Fault Injection Campaigns

sum up, QuickSort is the most resilient and available algorithm of the algorithms considered. The following summarizes the resilience-related observations, where the lower numbers are better (lower number of faults in those categories):

SDCs : Bubble < Quick < Radix < Merge < Heap

Segmentation Faults : Heap < Quick < Merge < Radix < Bubble

In addition to logging execution outcomes, we also maintained a mapping of the dynamic instruction where the fault was injected into the outcome that was produced in that execution. We draw some interesting observations from this mapping. In BubbleSort, half of the faults injected into registers that are employed in computing the index of the array access in the expression `Array[i-1]` produce segmentation faults. In HeapSort, injecting faults into the instructions executed just before and just after the recursive calls causes a high percentage (close to 75%) of SDC faults. Such precise profiling of fault injection sites enables us to observe critical instructions, specific to a sorting algorithm, where error injection leads to SDCs. Note that, as an area of future work, one can potentially extend the notion of such critical regions of an algorithm to other algorithms that follow similar design patterns. Furthermore, targeted fault detection and recovery mechanisms can be employed around these critical regions to provide cheap and effective means for improving resilience of programs to transient faults.

2.4 Discussion: Dynamic Analysis for Resilience

Fault injection at the source/intermediate representation (IR)-level has been a common approach to profiling the sensitivity of instructions to output quality. Fault injectors such as KULFI [1], LLFI [26], PDSFIS [29] promise accurate perturbation of instructions at runtime. These techniques perform their runtime analysis on instrumented source/IR operating on a particular concrete input or a set of inputs. Therefore, although these techniques achieve high levels of accuracy, there still need to be several thousands of fault injections to obtain statistically significant inferences (see Section 2.3.2), which may be cost-prohibitive.

Dynamic fault injection-based techniques are highly scalable and can be used to study system behavior under precisely emulated fault models. Fault injectors have been used on real-world HPC codes operating on large workloads [30, 31, 32] for targeted fault injection and analysis. Empirical fault injection and consequence analysis tools have been used on scientific applications to infer, with a fine granularity, sections of the application, such as data structures, etc., that are vulnerable to a particular class of errors [30]. Such a runtime methodology also provides the user with information about the temporal criticality

of a vulnerability of a specific application operating on a specific workload. Therefore, it is conclusive that such runtime fault injection techniques are effective in the process of associating the critical data objects to the vulnerabilities that corrupt their integrity, for large-scale applications.

Although dynamic analysis techniques are scalable and precise, they are not generalized since their inferences are bound to specific concrete executions on specific concrete workloads. Therefore, in order to gain statistically significant results, one has to perform multiple fault injections. This can be inefficient and unreliable since the parameters of the environment may not be reproducible. To mitigate these shortcomings, certain heuristics have been applied to provide accurate reasoning using a smaller subset of application behaviors. Offline dynamic analysis techniques [33, 34] provide information on dataflow and correlation difference. Dataflow difference ranks the criticality of instructions based on the number of inaccuracies that flow into it, whereas correlation differentiation performs a resilience profile based on correlating the number of unacceptable outputs to the number of fault injections. The former may be inaccurate as it is based on dataflow-based static analysis (we discuss this in detail in Section 3.5) and the latter does not provide formal guarantees on injected faults. Although there are optimizations for selective instruction perturbation [35], these techniques are limited by their ability to reason only about a subset of all the possible executions of the program.

In the next chapter, we provide a symbolic approach that performs a precise differential reasoning on all *divergent* behaviors of an application subject to inaccuracies, relative to its precise execution. We will show that this symbolic approach addresses the shortcomings of dynamic analysis-based approaches by reasoning about abstract symbolic states. Although symbolic approaches have their limitations with respect to scalability, precision, and applicability to scientific applications, they are accurate and customizable. It is intuitive to counteract the shortcomings of one approach with the other by employing a technique that uses both approaches in a symbiotic manner. The symbolic and dynamic machinery nicely complement each other, thus holding promise for future resilience studies.

CHAPTER 3

SYMBOLIC ANALYSIS FOR RESILIENCE

Our aim is to effectively facilitate automation of annotation-based programming practices for approximate and resilient computing. We require an approach that provides a formal reasoning about *all* approximated program behaviors, unlike the dynamic approaches outlined in the previous chapter which reason about a limited set of executions. Dynamic analyses reason about specific parameters of a concrete execution to determine the criticality of a program component. In order to provide formal guarantees, one has to consider a level of abstraction higher than a concrete execution. The abstraction is formalized by a relative specification between the precise and approximated program executions. A violation of this relative specification provides the notion of criticality to the given approximation. We intend to provide formal classification of computations driven by such relative specifications. To do this, we provide an approach that builds on differential static analysis techniques [36].

In this chapter, we will discuss our approach that computes a symbolic semantic difference between an application and its approximated version, to verify if the approximation satisfies a given relative specification. In this thesis, the semantic difference is geared towards control flow differencing to find computations whose relaxation influences control flow, i.e., we choose to model the relative specification in order to classify computations based on their control flow dependency. Previous studies [37] have reported that as much as 70% of the transient faults disturb program control flow. Control flow critical statements, when allowed to be approximate, typically lead to serious problems in guaranteeing program termination, unacceptably high corruptions in output data, and program crashes. By providing developers with a list of control flow critical statements in their program, we allow them to either inject appropriate error detection and correction mechanisms, or to avoid approximating the computations in the listed statements.

Figure 3.1 defines the syntax of our simple programming language, which is a subset of the Boogie language [38].

$$
\begin{aligned}
\mathit{Type} &::= \textbf{int} \mid [\textbf{int}]\textbf{int} \mid \textbf{bool} \\
\mathit{Program} &::= (\textbf{var}\ \mathit{Id} : \mathit{Type};)^*\ \mathit{Procedure}^+ \\
\mathit{Procedure} &::= \textbf{procedure}\ \mathit{Id}((\mathit{Id} : \mathit{Type})^{,*})\ \mathit{Returns}^?\ \{\mathit{Body}\} \\
\mathit{Returns} &::= \textbf{returns}(\mathit{Id} : \mathit{Type}) \\
\mathit{Body} &::= (\textbf{var}\ \mathit{Id} : \mathit{Type};)^*\ \mathit{BasicBlock}^+ \\
\mathit{BasicBlock} &::= \mathit{Label} :\ \mathit{Stmt};\ \textbf{goto}\ \mathit{Label}^{,+}; \\
&\quad \mid \mathit{Label} :\ \mathit{Stmt};\ \textbf{return}; \\
\mathit{Stmt} &::= \mathit{Stmt};\ \mathit{Stmt} \\
&\quad \mid \mathit{Id} := \mathit{Expr} \mid \mathit{Id}[\mathit{Id}] := \mathit{Id} \\
&\quad \mid \textbf{skip} \\
&\quad \mid \textbf{havoc}\ \mathit{Id} \\
&\quad \mid \textbf{call}\ \mathit{Id}(\mathit{Id}^{,*}) \mid \textbf{call}\ \mathit{Id} := \mathit{Id}(\mathit{Id}^{,*}) \\
&\quad \mid \textbf{assume}\ \mathit{Expr}
\end{aligned}
$$

Figure 3.1: The Syntax of our Simple Programming Language. *Id*, *Label*, and *Expr* have the usual meaning.

3.1 Preliminaries

The language supports integer type **int**, integer array type [**int**]**int**, and boolean type **bool**. A program in our language declares a set of global variables and a set of one or more procedures; there is one top-level entry procedure. A procedure has zero or more input parameters and can declare an output variable; its body contains local variable declarations and a nonempty set of basic blocks. Each basic block in a program is uniquely defined with its *Label*, and consists of a block statement *Stmt* followed by a control flow statement (either **goto** or **return**). We assume each procedure has a unique entry block. Program statements have their usual meaning. Statement **goto** $L_1, \ldots, L_n$ nondeterministically jumps to any one of the n labels, while the **return** statement returns from a procedure. Statement **havoc** x sets variable x to an arbitrary value, while the **call**-statement denotes a procedure invocation. The **assume** *Expr* statement proceeds only when *Expr* evaluates to true; otherwise, it blocks program execution. We use **assume**-statements in combination with **goto**-statements to model conditional branching. Detailed semantics of these statements can be found in the Boogie manual [39]. Figure 3.2 gives an example program in our simple programming language.

Assuming that all basic blocks in a program have unique basic block identifiers/labels L_i, we define an execution of a program as follows.

Definition 3.1 *An* execution Γ *of a program is a potentially infinite sequence of visited*

```
var x:int;

procedure foo(y:int) returns (r:int) {
L0:
  y := x + 3;
  y := y * y;
  x := x * y;
  goto L1,L2;

L1:
  assume x > 0;
  r := x + 5;
  goto L3;

L2:
  assume !(x > 0);
  r := x + y;
  goto L3;

L3:
  return;
}
```

Figure 3.2: Our Running Example Program in our Simple Programming Language.

basic blocks $L_0, L_1, \ldots$ *that is permissible by the language semantics. Such a sequence of basic blocks is representative of the control flow behavior of a particular program execution. A partial execution* Γ^n *is a finite prefix* $L_0, L_1, \ldots, L_n$ *of* Γ*. We use* $\Gamma(i)$ *to denote the* i*th element of* Γ*,* $\forall i \in [0, |\Gamma|]$*; otherwise,* $\Gamma(i)$ *is undefined.*

Next, we define the control flow equivalence between two executions.

Definition 3.2 *Program executions* Γ_1 *and* Γ_2 *are* control flow equivalent *if and only if* $\forall n \geq 0 \,.\, \Gamma_1(n) = \Gamma_2(n)$ *and* $\Gamma_1(n)$ *is defined iff* $\Gamma_2(n)$ *is defined.*

Note that two executions are said to be control flow equivalent if and only if they visit the same basic blocks in the same order of succession.

3.2 Capturing Control Flow Equivalence

Following the definitions of establishing a control flow footprint of program executions and their control flow equivalence, we define, in this section, the program transformations we propose to track control flow by storing a sequence of visited blocks. We also formally state what it means for a fault to be control flow critical.

3.2.1 Control Flow Tracking Using Arrays

To be able to capture control flow equivalence, we propose a source-to-source program transformation α that tracks control flow of a program execution as a part of the program's global state. Program transformation α takes a program as input and performs the following steps:

- It adds global variables *executionHistory* (a *ghost* array) and *count* (the respective counter). Note that *executionHistory* initially contains arbitrary *symbolic* values, while *count* is initialized to 0.
- It instruments every basic block of a program with a call to procedure $track(id : int)$ that appends unique basic block IDs to *executionHistory* in their order of execution.

Figure 3.3 is an example of program transformation α applied on our running example in Figure 3.2.

Proposition 3.1 *Program transformation α does not influence control flow of program executions.*

This simply follows from the observation that the transformation only adds ghost variables, statements, and procedure calls that do not affect control flow (also, no new basic blocks are added by these constructs).

For a partial program execution Γ^n, variable *count* after the execution of the last basic block $\Gamma^n(n)$ is equal to n. The updates to *executionHistory* follow the order of executed basic blocks. Given that *executionHistory* initially contains arbitrary values and *count* is initialized to 0, we inductively infer the following two propositions.

Proposition 3.2 *Program transformation α precisely captures control flow of a program execution in executionHistory.*

Proposition 3.3 *Control flow equivalence of partial executions is determined by comparing the values of executionHistory in the final basic blocks of the corresponding partial executions.*

```
var x:int;

// *** tracking begin ***
var executionHistory:[int]int;
var count:int;
procedure track(id:int) {
  executionHistory[count] := id;
  count := count + 1;
}
// ***  tracking end  ***

procedure foo(y:int) returns (r:int) {
L0:
  count := 0; // initialize count
  track(0); // tracking
  y := x + 3;
  y := y * y;
  x := x * y;
  goto L1, L2;
L1:
  track(1); // tracking
  assume x > 0;
  r := x + 5;
  goto L3;
L2:
  track(2); // tracking
  assume !(x > 0);
  r := x + y;
  goto L3;
L3:
  track(3); // tracking
  return;
}
```

Figure 3.3: Our Running Example with Injected Control Flow Tracking (denoted with *tracking*)

3.2.2 Fault Injection

Our main objective is to formally infer statements of a program that are control flow critical. If a statement under a fault affects control flow behavior, then it is control flow critical. We now discuss our approach of generating a faulty/relaxed program from its precise counterpart. The intention of the relaxed program is to formally model potentially erroneous or approximate computations so that we can analyze their effects on program executions. Currently, we assume an over-approximate fault model that potentially completely corrupts

a computation; that is, we consider the entire spectrum of approximations that are possible on a computation. We chose this model to provide strong guarantees about worst-case behavior of programs that operate under potentially faulty or approximate computations.

We define a source-to-source program transformation β that injects a fault into one of the statements of the input program P. The injected fault models a runtime error/approximation on the statement's computation by setting the result of the computation to an arbitrary value. We target simple assignment statements of the form $Id := Expr$ (see Figure 3.1) as candidates for fault injection in program P. We leverage the **havoc**-statement for fault injection, which assigns an arbitrary value to its argument. Such a statement is injected just after the chosen assignment, as illustrated in Figure 3.4.

Clearly, program transformation β used for fault injection preserves the control flow graph of the input program. Although the tracked control flow graph remains the same between the two program versions P and $\beta(P)$, it should be noted that the set of allowed executions of fault-injected program $\beta(P)$ may be different from the original program P.

Proposition 3.4 *If an execution Γ_1 of the original and Γ_2 of the fault-injected program starting from the same initial state have differing values of the array executionHistory in any*

```
var x:int;

procedure foo(y:int) returns (r:int) {
L0:
  y := x + 3;
  havoc y; // fault-injection
  y := y * y;
  x := x * y;
  goto L1, L2;
L1:
  assume x > 0;
  r := x + 5;
  goto L3;
L2:
  assume !(x > 0);
  r := x + y;
  goto L3;
L3:
  return;
}
```

Figure 3.4: Our Running Example with Fault Injected. The fault is injected into an assignment to variable y (denoted with *fault-injection*)

of their respective partial executions Γ_1^n *and* Γ_2^n *of the same length* n*, then the injected fault affects control flow behavior of the original program.*

It is given that the partial executions Γ_1^n and Γ_2^n have different values of the array storing sequences of basic blocks (*executionHistory*). We know that the control flow graph does not change due to the injected fault. Thus for the values of *executionHistory* to be different, the fault must have enforced a visitation order of basic blocks in Γ_2^n that is different from the sequence of basic blocks visited in Γ_2^n. That is $\exists i \in [0, n] : \Gamma_1(i) \neq \Gamma_2(i)$ and both basic block IDs are defined. Since both executions start from the same initial state, according to Definition 3.2, the injected fault affects control flow behavior of the original program. Note that a nondeterministic program is not control flow equivalent to itself. Therefore, sound conclusions about whether a fault affects control flow cannot be drawn from inherently nondeterministic programs.

The fault model that we consider in the transformation β mimics a completely unreliable hardware specification. We discuss more intricate fault models that we can use and a typical approach to encode them using Boogie in Section 5.2.

3.3 Verification Using Symbolic Differencing

In the previous section, we described our control flow tracking approach and fault-injection strategy, and we formalized properties of relaxed program executions that deviate in control flow compared to precise program executions. In this section, we define the notion of verifying control flow equivalence of a precise deterministic program and its relaxed version. We then formally define the verification problem and our approach to encode control flow equivalence with uninterpreted functions.

3.3.1 Verification Problem

Definition 3.3 *Let* D *be a deterministic program. Also, let* $P = \alpha(D)$ *be the precise and* $Q = \beta(\alpha(D))$ *the relaxed version of* D*. Then, programs* P *and* Q *are* control flow equivalent *if and only if for any initial state the two respective executions are control flow equivalent. In other words, the value of the respective executionHistory arrays is equal in both programs for all partial executions starting from the same arbitrary initial state.*

It follows from this definition and previously outlined propositions that verifying control flow equivalence suffices to prove that a fault does not affect control flow for any program input.

We encode this verification problem as an instance of symbolic program equivalence. We extend our language in Figure 3.1 with the **assert** *Expr* statement that reports a failure when *Expr* evaluates to false; otherwise, it acts as the **skip**-statement.

We compose programs P and Q into a single procedure *verifyCFEquivalence*() to assert their control flow equivalence. We illustrate (in Figure 3.5) the structure of this procedure by applying it to verify control flow equivalence of our running example (say P) and its imprecise counterpart (say Q).

The program Q is executed after its initial state x, y is saved. The control flow tracking variables *executionHistory* and *count* are empty and 0, respectively, to begin with. After P finishes execution, the final states of variables *executionHistory* and *count* are saved. This is the control flow footprint of program Q. After restoring the saved initial states to the

```
var x: int;
var executionHistory:[int]int;
var count:int;

// omitted tracking and fault-injection procedures

procedure verifyCFEquivalence(y:int) {
  var x0, x1, count1: int;
  var executionHistory0, executionHistory1: [int]int;

  // Save globals
  x0 := x;
  executionHistory0 := executionHistory;

  call r := foo_relaxed(y); // invoke relaxed procedure

  // save output globals
  executionHistory1 := executionHistory;
  count1 := count;

  // restore globals
  x := x0;
  executionHistory := executionHistory0;

  call r := foo_precise(y); // invoke precise procedure

  // check control flow equivalence
  assert(count1 == count && executionHistory1 == executionHistory);
}
```

Figure 3.5: Encoding of Control Flow Equivalence as Symbolic Program Equivalence

variables x, y and reinitializing the control flow tracking variables, the control flow footprint of program P is generated. The procedure then asserts the equality of the control flow footprints of P and Q. Verification is performed by searching for an input state for which the assertion fails, failing which the programs are control flow equivalent.

This algorithm of verification guarantees a partial control flow equivalence guarantee. It is sound and complete for terminating executions, i.e., if there is a control flow deviation, then the terminating execution will violate the assertion and vice versa. However, for nonterminating executions, it is not the case that the assertion fails whenever there is a control flow deviation. The checking of the assertion is contingent on the programs terminating on the input.

3.3.2 Verification Using Uninterpreted Functions

While composed program verification by comparing the history of visited basic blocks is sound and complete for terminating executions of deterministic programs, it is not efficient for the underlying theorem prover that verifies the assertion. SMT solvers have an inherent complexity blow-up when it comes to handling the theory of arrays, among a multitude of theories employed to prove the relative correctness specifications. We also observe this increase in complexity in our empirical case study. To handle this verification of control flow equivalence of program versions efficiently, we propose a technique that employs uninterpreted functions (in short, UIF). *Expr* in the grammar in Figure 3.1 allows for specification of uninterpreted function symbols. Our aim is to develop a formalism that subsumes the purpose of the global array, *executionHistory*, to track control flow history of program executions, whilst being more efficient. The optimization brought in by this technique reduces the verification of control flow equivalence from comparing basic block sequence arrays to comparing a single value that depends on the entire history of visited basic blocks.

We propose a program transformation γ that essentially captures the control flow signature of a program execution by leveraging an uninterpreted function. Program transformation γ performs the following operations:

- Augments the global state with the integer *executionSummary*.
- Instruments the basic blocks of the program with a call to an uninterpreted function $trackUIF(executionSummary, blockID)$
- *executionSummary* is updated with a unique value which is a function of *executionSummary* and *blockID* in the order of execution of basic blocks.

Note that *executionSummary* is assigned the same uninterpreted value in the initial global state of the program execution. Figure 3.6 is an example of program transformation γ applied on our running example in Figure 3.2

It is clear, using arguments similar to Proposition 3.1, that the program transformation γ does not influence the control flow behavior of the source program. In order to prove that program transformation γ can effectively replace α in verifying control flow equivalence, we need to establish that uninterpreted functions have the same control flow tracking semantics as sequences of executed basic blocks.

Theorem 3.1 *Two partial executions* Γ_1 *and* Γ_2 *(of* P *and* Q *respectively) are control flow equivalent if and only the values of executionSummary are identical at the end of the two*

```
var x:int;
var executionSummary:int;

function trackUIF(summary:int, id:int): int;

procedure foo(y:int) returns (r:int) {
L0:
  executionSummary := trackUIF(executionSummary, 0);
  y := x + 3;
  y := y * y;
  x := x * y;
  goto L1, L2;

L1:
  executionSummary := trackUIF(executionSummary, 1);
  assume x > 0;
  r := x + 5;
  goto L3;

L2:
  executionSummary := trackUIF(executionSummary, 2);
  assume !(x > 0);
  r := x + y;
  goto L3;

L3:
  executionSummary := trackUIF(executionSummary, 3);
  return;
}
```

Figure 3.6: Our Running Example with Injected Control Flow Tracking UIFs

executions.

Proof. We only provide a proof sketch here. First, consider the case when Γ_1 and Γ_2 have different control flows; i.e., either they have different lengths, or they differ in some position. In either case, consider the interpretation of *trackUIF* function where it assigns every distinct tuple of its argument to a distinct value; this is a feasible interpretation as *trackUIF* is completely unconstrained. It is easy to see that the final values of *executionSummary* will be different in the two executions for such an interpretation of *trackUIF*. On the other hand, if control flows are identical in Γ_1 and Γ_2, then the final values of *executionSummary* are identical since *trackUIF* is a function.

With this equivalence of *executionSummary* and *executionHistory* established, the definition of the verification problem can be restated as in Definition 3.4.

Definition 3.4 *Let D be a deterministic boogie program. Let $P = \alpha(D)$ be the precise and $Q = \beta(\alpha(D))$ be the relaxed version of Q. P and P' are control flow equivalent iff there is* no *initial state for which P and Q have program executions, Γ and Γ', that differ in control flow behavior, i.e., the value of executionSummary is the same for all finite prefixes of Γ and Γ' that begin executing from the same arbitrary initial state.*

3.4 Implementation

Based on the introduced techniques, we implemented a prototype tool flow for computing the set of control flow critical computations from applications written in C. Figure 3.7 shows our tool flow. First, HAVOC [40] translates a C program into its corresponding Boogie program. HAVOC uses Boogie arrays to model the heap of a C program, where pointer and field dereferences are modeled as array reads and writes. We use a modified version of HAVOC that incorporates deterministic modeling of dynamic memory allocation [41].

Next, we automatically inject control flow tracking into the output program of HAVOC to create two program copies (see Sections 3.2.1 and 3.3.2), and we inject a fault into one of them (see Section 3.2.2). Loops and recursion in the two programs are now unrolled to a user-specified depth, and all procedure calls are inlined. The formal guarantees we currently provide are hence sound and complete only up to the particular unroll depth. We discuss a sound proof checking technique, which we are currently working on, to avoid loop unrolling in Section 5.2.

The inlined programs are verified for control flow equivalence using the SymDIFF tool [42] as described in Section 3.3.1. SymDIFF uses the Satisfiability Modulo Theories (SMT) solver Z3 [43] for discharging the final control flow tracking assertion. Violations of the control flow

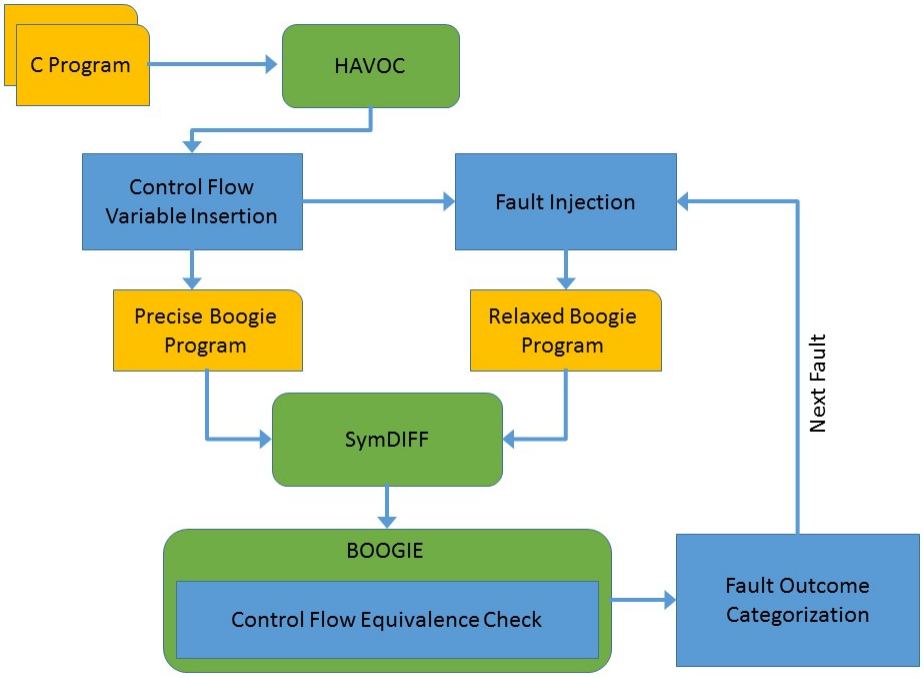

Figure 3.7: Our Tool Flow for Profiling Control Flow Resilience.

equivalence specification are captured by the counterexample traces that SymDIFF produces: if a counterexample is generated, the fault-injected instruction is control flow critical, and vice versa. We repeat this process for all potential fault sites of the analyzed program to obtain a complete resilience profile with respect to fault/approximations that influence control flow.

3.5 Experiments

We performed the evaluation of our symbolic resilience profiler on several benchmarks written in C, including sorting, image processing [44], data structure implementations, and operations on matrices. Specifically, the benchmarks (details in Table 3.1) are:

- *InsertionSort, BubbleSort, SelectionSort* are standard implementations of the corresponding sorting algorithms;
- *Brightness Correction* is used to enhance the visual appearance of an image;
- *Arithmetic Mean Filter* operation on an image removes short tailed noise such as uniform and Gaussian type noise from the image at the cost of blurring the image;
- *Centroid Computation* is an algorithm to compute the centroid of the object within a given image;

Table 3.1: Characteristics of our benchmarks. LOC and #Procs are the number of lines and the number of procedures in the sourcefile, respectively; #Faults is the total number of potential fault sites.

Benchmark	LOC	#Procs.	#Faults
Bubble Sort	25	2	13
Selection Sort	30	2	15
Insertion Sort	24	2	13
Brightness Correction	21	1	8
Centroid Computation	55	3	30
Arithmetic Mean Filter	27	1	13
Linked List Operations	76	5	40
Array Map Operations	78	7	35
Matrix Multiplication	38	3	17

- *Matrix Multiplication* performs multiplication of two square matrices using the naive algorithm;
- *Linked List Operations* contains basic modules to create, modify, find, and delete elements of a linked list data structure;
- *Array Map Operations* maps a function over the values of an array and updates it.

Table 3.2 gives experimental results and comparison of our proposed control flow resilience profiling techniques.

Clearly, control flow tracking using an uninterpreted function outperforms the naive encoding using an array, which is not surprising given often inefficient handling of the theory

Table 3.2: Experimental results and comparison of our control flow resilience profiling techniques. Experimental results and comparison of our control flow resilience profiling techniques. #Faults is the total number of potential fault sites; #CF-Equiv. is the reported number of faults that do not affect control flow; Time(Array) is runtime of the array-based control flow tracking; Time(UIF) is runtime of the UIF-based control flow tracking. All runtimes are in minutes and timeout per fault is 30 minutes.

Benchmark	#Faults	#CF-Equiv.	Time(Array)	Time(UIF)
InsertionSort	13	1	2.8	1.3
BubbleSort	13	1	2.3	1.6
SelectionSort	15	2	3.7	1.8
Brightness Correction	8	4	1.3	1.4
Arithmetic Mean Filer	13	5	2.6	1.3
Centroid Computation	30	14	229.1	8.3
Matrix Multiplication	17	7	180.0	5.4
Linked List Operations	40	7	231.7	55.4
Array Map Operations	35	12	960.9	115.4

of arrays by SMT solvers. In fact, we observe several timeouts in the case of *Array Map Operations* and *Linked List Operations* when using array-based instrumentation. We safely classify such outcomes as faults that can induce a control flow deviation.

Given such resilience profiles, we can discuss which applications are more amenable to approximate computation. For example, applications in the domain of image processing are more tunable to approximations that do not affect control flow, since most computations are local to a pixel neighborhood. Our technique automatically identifies statements whose computation can be approximated without affecting control flow as candidates for safe approximations. For example, *Centroid Computation* depends on the area of an object within an image. Although the instructions to compute the area are critical to control flow, the resultant value of the area does not affect control flow behavior. The value of the area is read multiple times for computing the centroid within a loop of high complexity, and since our tool determines it does not affect control flow, one can annotate it to be stored in approximate memory (e.g., with low supply-voltage/refresh rate) for energy savings.

In general, one may assume that taint analysis is sufficient to precisely verify control flow equivalence in the presence of hardware inaccuracies. This assumption is invalid when the injected inaccuracy is semantically independent from the control flow behavior of the program. For example, consider the injected relaxation in variable y in Figure 3.3. The control flow path taken is determined by the sign of variable x, since the value of y is always nonnegative. Therefore, there is no input state for which the two program versions take different control flow paths. This is accurately noted by our technique and does not produce a verification condition violation. However, traditional information flow-based taint propagation rules will flag that the variable x is tainted and therefore the control flow paths can be arbitrary. It is this level of precision that we aim to achieve to target control flow-critical applications, and our experiments show promise in such techniques for profiling application resilience.

We performed a comparison of our control flow profiling technique with a taint analysis technique available in the latest version of the SymDIFF release [45]. The results of this taint analysis are available in Table 3.3.

We can see that the taint analysis is much faster than our approach, especially in the benchmark instances such as centroid computation, array map operations, linked list operations, and matrix multiplication. Although our technique is slower, it has the advantage of being more precise than taint analysis. For the benchmark instances of selection sort, linked list operations, and array map operations, the taint analysis proves to be conservative and therefore over-taints the set of approximable computations. This is due to the fact discussed before that when the semantics of the statement determine the control flow path of the

Table 3.3: Control flow profiling with taint analysis. Runtimes are in minutes.

Benchmark	#Faults	#CF-Equiv(Taint).	Time(Taint)
InsertionSort	13	1	1.1
BubbleSort	13	1	0.8
SelectionSort	15	1	1.9
Brightness Correction	8	4	0.4
Arithmetic Mean Filer	13	5	2.5
Centroid Computation	30	14	3.3
Matrix Multiplication	17	7	2.9
Linked List Operations	40	2	0.8
Array Map Operations	35	3	2.5

program, taint analysis can tend to generalize its conclusions. For example, selection sort is an interesting benchmark where traditional taint analysis flags a control flow independent statement as unsafe. The selection sort algorithm sorts an array A of length n by pushing the maximum element of the subarray $A[0 \ldots i-1]$ to the position i after every iteration. Therefore, the invariant of this algorithm is that once an element has been pushed to the end, it will never play a part in determining control flow behavior. Therefore, our technique identifies that the instruction that pushes the maximum element to the end of the array at the end of every iteration can be annotated to store the value in an approximate memory region. Traditional taint propagation rules will flag the whole array object as tainted after the annotation of this instruction to be approximate. Therefore, such taint analyses may conclude that the control flow behavior will change because the array on which the algorithm operates is tainted. Verification of this well-known invariant of the selection sort algorithm reinforces that our techniques are sound and complete for bounded nonblocking executions of deterministic programs.

CHAPTER 4

RELATED WORK

In this section we explore previous work related to the different focal points of this thesis, such as algorithmic resilience, fault injection, control flow tracking, program annatations and reliability proofs.

1. **Algorithmic Resilience:** Algorithm level fault detection can be studied by focusing either on memory faults or computational faults. Sorting algorithms and data structure resilience have recently been studied focusing on a DRAM fault model, where faults are assumed to occur only on the algorithm inputs [28, 27]. In this work, various algorithms are compared based on the k-unsortedness metric. More specifically, the lower the number of misplaced data items, the more resilient the algorithm is deemed to be. In contrast, in our case study we assume a more fine grained fault model that accommodates more fault categories, specifically, at the register and control flow level. We also compare algorithms based on the number of silent data corruptions.

2. **Fault Injection:** One of our main contributions is the development of a fault injector called KULFI, based on the LLVM compilation infrastructure [22, 23]. KULFI can inject transient faults into a chosen data register of a randomly chosen program instruction at run time. Several previous studies have exploited fault injectors similar to KULFI [46, 47]. There are also efforts that directly inject faults into the hardware. Hardware-based fault injection is less flexible [48] and not as programmable. Fault injectors can also be built by exploiting OS-level facilities [46, 47]. Other software-level fault injectors include those based on PIN [29, 25]. Specifically, the PDSFIS fault injector [29] uses Intel's PIN framework.

   In contrast to the above works, KULFI uses the open-source LLVM compiler infrastructure, similarly to other recently reported fault injectors [25, 26]. The fault injector LLFI [26] is primarily geared towards injecting errors in soft-computing applications. LLFI and KULFI were developed concurrently, and currently they share many similar

features. However, when we started working on this project, KULFI was the only tool available to us that had all of the required features. For example, KULFI provides fine-grained error injection control (briefly discussed in the next section), which suits well our requirements for performing the empirical evaluations described in this paper. The fault injector used in the Relax framework [25] also uses the LLVM compiler infrastructure. However, this fault injector is not publicly available. Furthermore, an informal study by a Relax user suggests that KULFI is easier to control and fine-tune, while also providing interesting command-line options not found in Relax [49].

3. **Control Flow Tracking:** Previous work [50, 51] has shown the applicability of control flow signatures in tracking invalid control flow graph transitions. The techniques boil down to computing control flow signatures on-the-fly at each node of the control flow graph when the program counter transfers into it. Our approach to tracking control flow behavior is based on static differential analysis and is employed predeployment of the application, whereas these techniques are useful in detecting and correcting control flow divergences at runtime.

4. **Annotating Programs for Approximations:** Recent years have seen the emergence of frameworks that enable programmers to perform relability engineering of their source code. EnerJ [11] is a type qualifier-based system for annotating data and computation as approximate. Rely [10] is another framework that checks programmer-relaxed versions of programs against a hardware reliability specification for quantifying error, while Relax [52] allows specification of code-blocks to be approximable. Much effort that follows this line of research has been to alleviate the burden of manual annotation from the programmer, so that such annotation-based frameworks can be utilized effectively. These efforts can be broadly classified into 2 categories: dynamic analysis-based approaches and approaches that aim to prove relaxed reliability.

5. **Proving Relaxed Reliability:** Recent research has focused on proving reliability properties of relaxed programs. An initial effort in this direction has been to provide a static quantitative reliability analysis for sound and verified reliability engineering [10]. This work builds on the Rely framework to prove, by performing a procedural precondition checking, if a particular relaxed program would guarantee an expected probability of output correctness. ExPAX [53] is a framework that generates a set of safe-to-approximate operations based on a dataflow-based taint analysis. It then uses a genetic optimization algorithm to compute the level of approximation for each

operation in the set so that energy consumption is minimized and reliability constraints are satisfied. Similarly, there have been extensions to the EnerJ framework [54] to accommodate annotation of reliability requirements in the source code. A user can specify, for each instruction in a subset of instructions, a probability that it is correct. Then based on the requirements on the output quality, the approximation levels of other instructions are determined. These techniques have the common objective of tuning the level of approximation of each computation to guarantee a particular *quantitative* reliability on the output of a single program. Our technique aims to provide formal guarantees on the reliability of a program in relative terms of the precise version rather than an absolute quantity. Carbin et al. [55] present language constructs for developing and specifying relaxed programs. Specifically, they provide proof rules for reasoning about relational acceptability properties and unary acceptability properties. Though this work also provides formal guarantees about relative program properties, our work focuses on formally detecting relaxations that cause a control-flow difference. We have also provided a tool-flow to verify relative properties of programs written in C, unlike this work that enforces one to specify programs in Coq. In our future work, we aim to make this differencing language-agnostic by working on programs written in the intermediate verification language Boogie.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This thesis provides two main orthogonal contributions to profile application resilience (a dynamic runtime analysis and a static symbolic control flow analysis) in support of the thesis statement. The techniques presented in this thesis showcase two fundamentally different pespectives to profile resilience and the trade-offs they establish, in terms of efficiency, accuracy and scalability.

Firstly, we presented a unique, thorough case study of resilience of several popular and widely used sorting algorithms to hardware faults. To be able to perform such an extensive resilience study, we first implemented a new, open source LLVM-level fault injector called KULFI. Faults injected by KULFI at the LLVM level provide a reasonable fault model for actual transient faults in the hardware. Using KULFI, we performed an extensive empirical study that observed behavior of sorting algorithms when faults are being injected. Based on the statistically significant results of this study, we drew informative conclusions about resilience of these algorithms and specifically obtained a profile of the criticality of instructions in the perspective of resilience.

We also proposed a symbolic technique to profile application resilience based on the relative correctness specification of control flow equivalence, under arbitrary faults in the computation. Specifically, we formalized the notion of control flow equivalence of precise and relaxed program versions in the presence of faults. We presented two approaches to track control flow behavior of programs written in Boogie. Our experimental results showed the applicability of the techniques on representative benchmarks. The runtimes showcase the usage of uninterpreted functions in efficient control flow equivalence proofs without the loss of expressive power. We also outlined the higher precision provided by our approach in comparison to traditional taint analysis-based approaches, for proving control flow equivalence. In summary, we have established the correctness of our thesis statement by the experimental results and the theories presented in this document.

5.2 Future Work

1. **Dynamic Analysis with KULFI:** As discussed in Section 2.4, although dynamic analysis-based approaches can be very fine-grained, they may prove to be inefficient and unsound. The basic shortcoming of such approaches is the limited guarantee of code coverage that they provide. A future direction for KULFI would be to include strategies for generating a representative set of inputs by code coverage analyses. Another well motivated extension to KULFI would be to support parallel applications (involving MPI, PThread, OpenMP, etc.) in order to broaden its use in the domain of high performance computing. KULFI currently injects errors with a instruction-level granularity. Moving to a more generic code-region based error injection strategies would improve the usability of KULFI.

2. **Symbolic Differencing for Resilience:** There are four areas of future work: fault models, relative assertions, scaling composed program verification and supporting other data types. Firstly, we intend to move ahead with more intricate fault models. Instead of corrupting the computation to an arbitrary value using the $havoc(x)$ statement, we will use the *assume* statement to enforce the corruption to satisfy a particular predicate. For example, by succeeding the $havoc(x)$ statement with $assume(-10 \leq x \leq 10)$, we control the corruption to take values in the interval $[-10, 10]$. Such constrained faults will really distinguish a semantics based approach, such as ours, from taint-analysis which typically tracks flow of information between computations.

   Likewise, we plan to explore a richer set of relative assertions in the composed program verification discussed in Section 3.3. For example, instead of placing an assertion on the control flow tracking variables, we can specify a relative assertion such as $assert(x_P - 5 \leq x_Q \leq x_P + 5)$ to verify if x_Q, the output of x in the relaxed program version Q, is within 5 units of x_P, the output of x in the precise program version P. We believe that the framework of differential assertion checking will be useful to discharge such proof obligations. The predicate specification language used in assert and assume statements spans logical expressions, basic arithmetic operations, relational operations and quantification over program variables. Thus interesting relative program specifications, formalized in previous work [56], can be written and verified using the same composed verification technique. Resilience can thus be profiled based on these relative specifications.

   The two main bottlenecks of efficiency when it comes to the composed program verification are unnecessary theorem prover calls and loops/recursion. The former

occurs when fault sites that obviously violate control flow property are still checked by the theorem prover for control flow violations. We can use taint propagation rules and analysis from previous work [45, 57, 58] to prune such obvious cases and reduce the number of theorem prover calls. For the latter, we are currently working on customizing a sound proof checker for control flow equivalence which does not require for the loops/recursion to be unrolled/inlined to a particular depth. This proof checker obviates the need to unroll loops by means of generating loop invariants (concise summaries of the loop executions).

Currently, our tool flow does not support programs that operate on floating points and bit vectors. The challenge lies in precisely encoding the semantics of floating point and bit vector computations, and is one that we will pursue in the future.

REFERENCES

[1] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *19th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2013, pp. 41–50.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[3] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 76:1–76:11.

[4] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 22:1–22:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503257

[5] P. N. Sanda, J. W. Kellington, P. Kudva, R. N. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, "Soft-error resilience of the IBM POWER6 processor," in *IBM Journal of Research and Development*, 2008, pp. 275–284.

[6] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75–82, 1997.

[7] J. A. Rivers, M. S. Gupta, J. Shin, P. N. Kudva, and P. Bose, "Error tolerance in server class processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 945–959, 2011.

[8] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 2013, pp. 1–6.

[9] S. Natarajan, *Imprecise and Approximate Computation*. Springer, 1995.

[10] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013, pp. 33–52.

[11] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 164–174.

[12] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, pp. 213–224, 2012.

[13] "KULFI: An instruction level fault injector," http://github.com/soarlab/KULFI/.

[14] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[15] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations*, 2014.

[16] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems (TOCS)*, pp. 222–238, 1983.

[17] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, pp. 200–209, 1962.

[18] M. Treaster, "A survey of fault-tolerance and fault-recovery techniques in parallel systems," *arXiv preprint cs/0501002*, 2005.

[19] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2007, pp. 1442–1447.

[20] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Softw. Eng.*, pp. 166–182, 1990.

[21] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 472–481.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[23] "The LLVM compiler infrastructure," http://llvm.org/.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[25] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 497–508.

[26] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," *Silicon Errors in Logic System Effects (SELSE)*, 2013.

[27] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, "Experimental study of resilient algorithms and data structures," in *International Conference on Experimental Algorithms*, 2010, pp. 1–12.

[28] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, "The price of resiliency: A case study on sorting with memory faults," in *Algorithmica*, vol. 53, no. 4, 2009, pp. 597–620.

[29] A. Jin, J. Jiang, J. Hu, and J. Lou, "A PIN-based dynamic software fault injection system," in *International Conference for Young Computer Scientists (ICYCS)*, 2008, pp. 2160–2167.

[30] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 57:1–57:11.

[31] C. Constantinescu, "Teraflops supercomputer: Architecture and validation of the fault tolerance mechanisms," *IEEE Transactions on Computers*, pp. 886–894, 2000.

[32] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, J. J. Beahan, A. Shamilian, and A. Nikora, "Fault injection experiment results in space borne parallel application programs," in *IEEE Aerospace Conference Proceedings*, vol. 5, 2002, pp. 5–2133.

[33] M. F. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Dynamic analysis of approximate program quality," Technical Report UW-CSE-14-03-01, University of Washington., Tech. Rep.

[34] M. F. Ringenburg, A. Sampson, L. Ceze, and D. Grossman, "Profiling and autotuning for energy-aware approximate programming," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[35] P. Roy, R. Ray, C. Wang, and W.-F. Wong, "ASAC: automatic sensitivity analysis for approximate computing," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, 2014, pp. 95–104.

[36] S. K. Lahiri, K. Vaswani, and C. A. Hoare, "Differential static analysis: Opportunities, applications, and challenges," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FOSER)*, 2010, pp. 201–204.

[37] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium (IOLTS)*, 2003, pp. 137–143.

[38] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, 2006, pp. 364–387.

[39] K. R. M. Leino, "This is Boogie 2," *Manuscript KRML*, vol. 178, p. 131, 2008.

[40] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer, "Unifying type checking and property checking for low-level code," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, pp. 302–314.

[41] M. Kawaguchi, S. K. Lahiri, and H. Rebêlo, "Conditional equivalence," *Microsoft, MSR-TR-2010-119, Tech. Rep*, 2010.

[42] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, 2012, pp. 712–717.

[43] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*, 2008, pp. 337–340.

[44] H. R. Myler and A. R. Weeks, *The pocket handbook of image processing algorithms in C*. Prentice Hall Press, 2009.

[45] "SymDiff: A platform for differential program verification." [Online]. Available: https://symdiff.codeplex.com/

[46] V. Sieh, "Fault-injector using UNIX ptrace interface," in *Internal Report 11/93, IMMD3, Universität Erlangen Nürnberg*, 1993.

[47] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995.

[48] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[49] S. Chen, personal communication, 2013.

[50] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *ACM SIGPLAN Notices*, vol. 48, no. 5. ACM, 2013, pp. 3–12.

[51] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 111–122, 2002.

[52] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 497–508.

[53] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "ExpAX: A framework for automating approximate programming," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[54] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Tuning approximate computations with constraint-based type inference," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[55] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 169–180.

[56] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Towards modularly comparing programs using automated theorem provers," in *International Conference on Automated Deduction (CADE)*, 2013.

[57] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis." in *Usenix Security*, 2005, pp. 18–18.

[58] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *NDSS*, 2007.